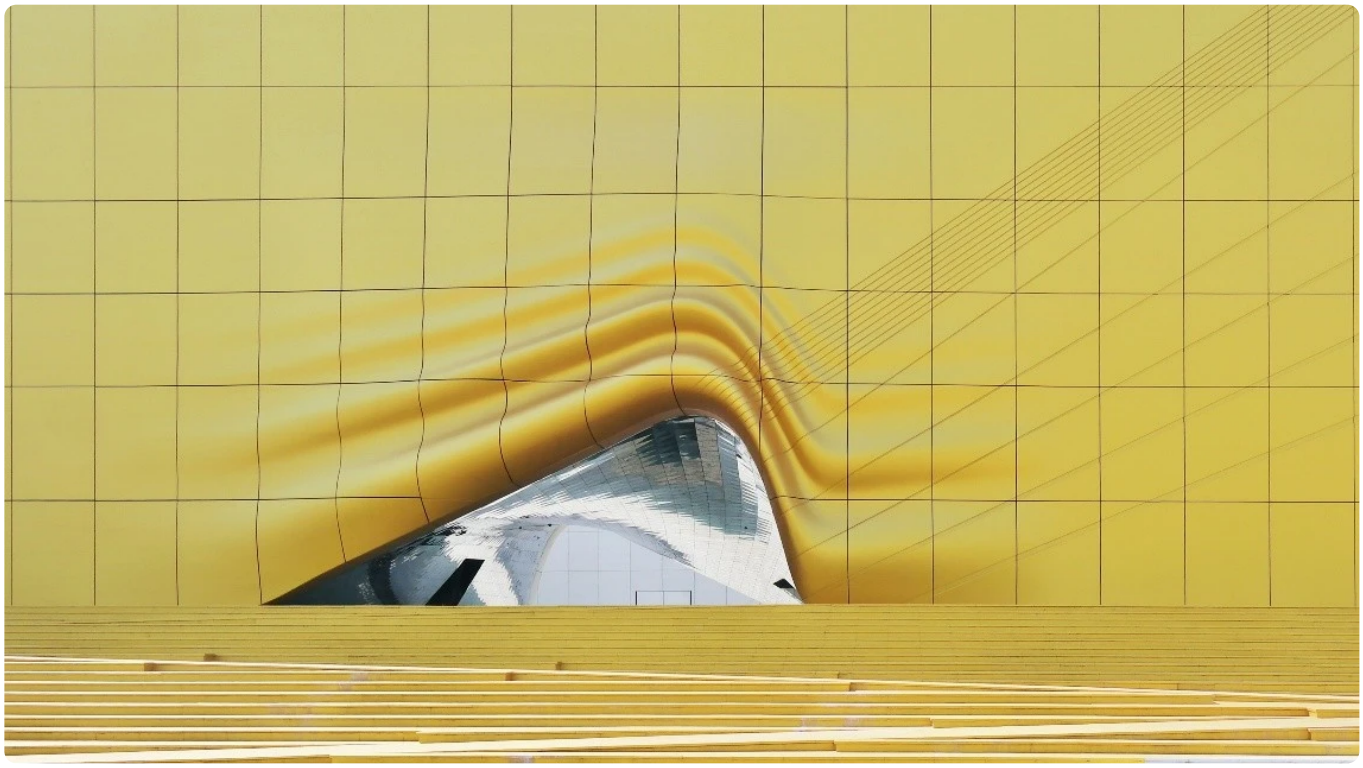
下载APP

12 | 分布式锁Chubby（一）：交易之前先签合同

2021-10-15 徐文浩

《大数据经典论文解读》 课程介绍 >



讲述：徐文浩

时长 18:17 大小 16.76M

你好，我是徐文浩。

在过去的十几讲课程里，我带你一起学习完了 GFS、MapReduce，以及 Bigtable 这三篇被称之为 Google 的“三驾马车”的论文。不知道你有没有发现，这三篇论文有一个共同点，那就是这三个系统**都是一个单 Master 系统**。而这就带来了一个问题，就是这个 Master 会成为整个系统的单点，一旦 Master 出现硬件故障，或者遇到 Master 网络不通的情况，整个集群就不能提供完整的服务了。

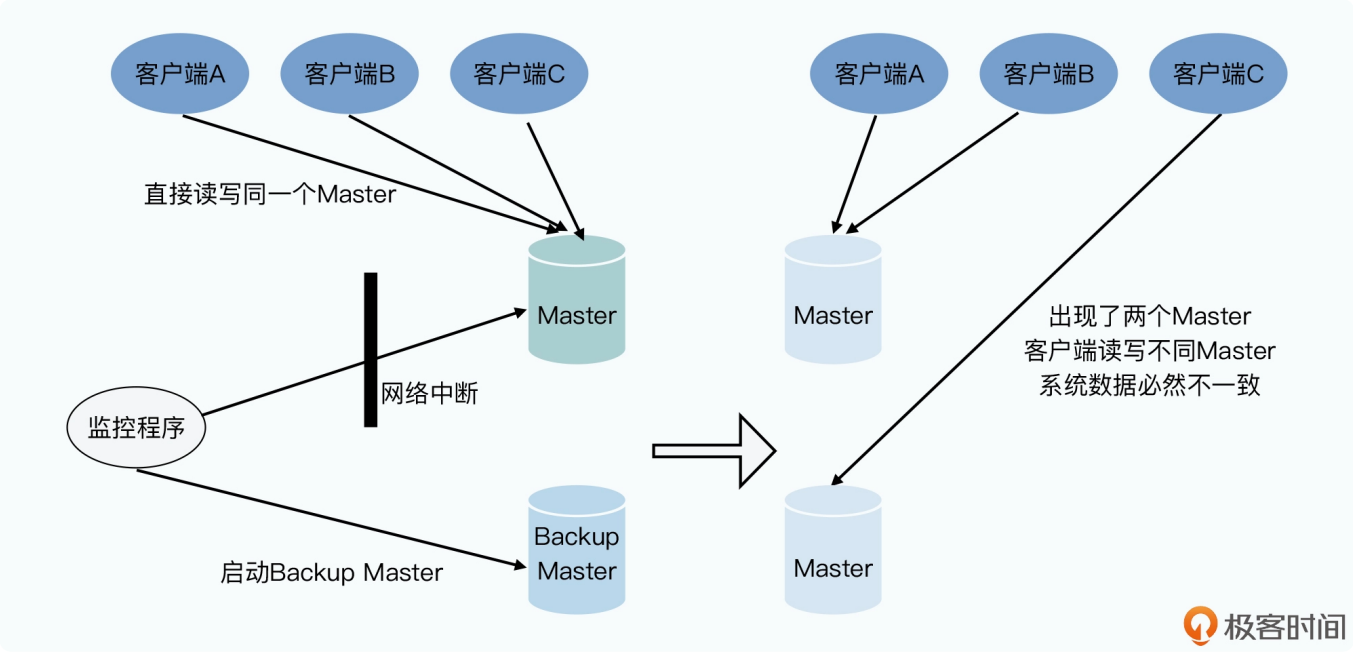
MapReduce 里的 Master 我们可以暂且不论，毕竟这个 Master 的生命周期只是一个 MapReduce 任务执行的时间，即使 Master 出了问题，简单地重跑一下任务就好了。但是 GFS 和 Bigtable，都是要长时间提供在线服务的系统。从概率的角度来讲，它们的 Master 也一定会遇到故障。

所以，在 GFS 和 Bigtable 的论文里，我们看到它们都有对应的 Backup Master 机制。通过一个监控机制，当发现 Master 出现问题的时候，就自动切换到数据和 Master 完全同步的 Backup Master，作为系统的“灾难恢复”机制。

乍一听，这个做法简单直接。不过如果仔细想一想，这个操作可没有那么容易实现。我们至少会遇到两个问题：

第一个，是我们怎么能够做到 Backup Master 和 Master 完全同步？特别是当硬件、网络可能出现故障的情况下，我们怎么能够做到两边的数据始终同步。如果数据不能做到始终同步，那么当真有需要我们切换节点到 Backup Master 的时候，我们就会遇到数据丢失的情况。

第二个，是监控程序本身也是一个单点，当我们的监控程序说 Master 挂了的时候，我们怎么知道 Master 是真的挂了，还只是监控程序到 Master 的网络中断了呢？如果是后者的话，会不会出现一个集群里有两个 Master 的情况？



监控程序和原先Master之间出现网络中断，就会出现多个Master，最终导致数据不一致

而这些问题的本质，就是我们接下来要讲解的**分布式共识问题**。并且这个分布式共识问题的解决，最终会落地成为 **Chubby** 这个粗粒度的分布式锁方案。我会分三个部分来讲解这个问题：

第一部分，是对于分布式共识问题的探讨，我会讲解二阶段提交和三阶段提交。通过这个过程，带你理解分布式一致性，以及 CAP 三者共存的挑战。这部分是一个基础知识，

你会开始对分布式一致性和分布式事务有一个入门性质的认识。

第二部分，我们会仔细来聊一下事务里的 ACID 到底是指什么，并且专门讲解一下 Paxos 算法，最后我们会一起来看看“可线性化”这个概念。理解了这些问题，你对分布式一致性和分布式事务就能有一个深入的理解了。

第三部分，则是对于《The Chubby lock service for loosely-coupled distributed system》的深入讲解，看看 Chubby 这个系统为什么这么设计，以及在大数据系统下的应用场景。

今天这一讲，我们主要来学习第一部分。相信通过这一讲的学习，你会理解到分布式系统难在哪里，以及对于 CAP 这三者无法同时满足这一点，有一个切身的体会。

从两阶段提交到 CAP 问题

在 GFS 的论文里，我们看到，GFS 的 Master 是有一个**同步复制**的 Backup Master 的。所有在 Master 上的操作，都要同步在 Backup Master 上写入成功之后，才算真正写入完成。这句话说起来很容易，可是实际上并不容易做到。

因为同步复制要求下的数据写入操作，要跨越两个服务器。所以我们不能像前面 Bigtable 里的 SSTable 那样，只要预写日志（WAL）写入成功，就认为在 Master 上数据写入成功了。因为很有可能，同步在 Backup Master 里写入的数据，会由于硬件问题或者进程忽然被 kill 等原因失败了。这个时候，所谓的“同步复制”也就不复存在了。

所以，让 Backup Master 和 Master 做到同步复制，本质上我们每次的成功就是一个分布式事务，也就是要么同时在 Master 和 Backup Master 上成功，要么同时失败。

为了解决这个分布式事务问题，我们需要有一个机制，使得 Master 和 Backup Master 两边的数据写入可以互相协同。那么第一个被想到的解决办法，就是**两阶段提交**（2PC，Two Phases Commit）。

两阶段提交的过程其实非常直观，就是把数据的写入，拆分成了提交请求和提交执行这两个不同的阶段，然后通过一个协调者（Coordinator）来协调我们的 Master 和 Backup Master。这个过程是这样的：

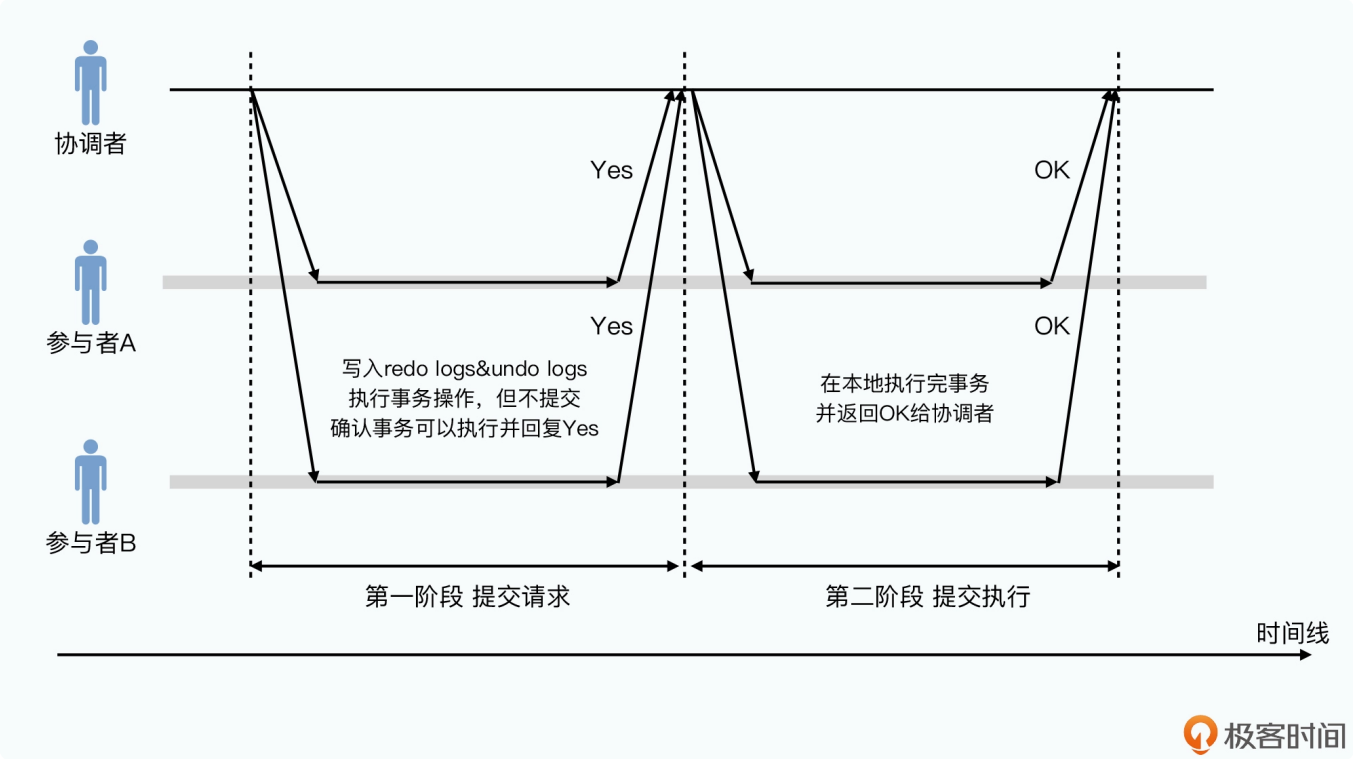
第一个阶段是**提交请求**。

协调者会把要提交的事务请求发给所有参与者，所有的参与者要判断自己是否可以执行这个请求，如果不行的话，它会直接返回给协调者，说自己不能执行这个事务。而如果它确定自己可以执行事务，那么，它会先把要进行的事务以预写日志的方式写入下来。

需要注意，这个写入和我们在 Bigtable 中所说的，写入日志就意味着数据写入成功有所不同。在提交请求阶段写入的 WAL 日志，还没有真正在参与者这里生效。并且，在写入的日志里，不仅有如何执行事务的日志（redo logs），也有如何放弃事务，进行回滚的日志（undo logs）。当参与者确定自己会执行事务，并且对应的 WAL 写入完成之后，它会返回响应给协调者说，“我答应你我会执行事务的”。

第二个阶段是**提交执行**。

当协调者收到各个参与者的返回结果之后，如果所有人都说它们答应执行这个事务。那么，协调者就可以再次发起请求，告诉大家，可以正式执行刚才的那个事务了。等实际的事务执行完成之后，参与者就会反馈给协调者，而协调者收到所有参与者成功完成的消息之后，整个事务就成功结束。

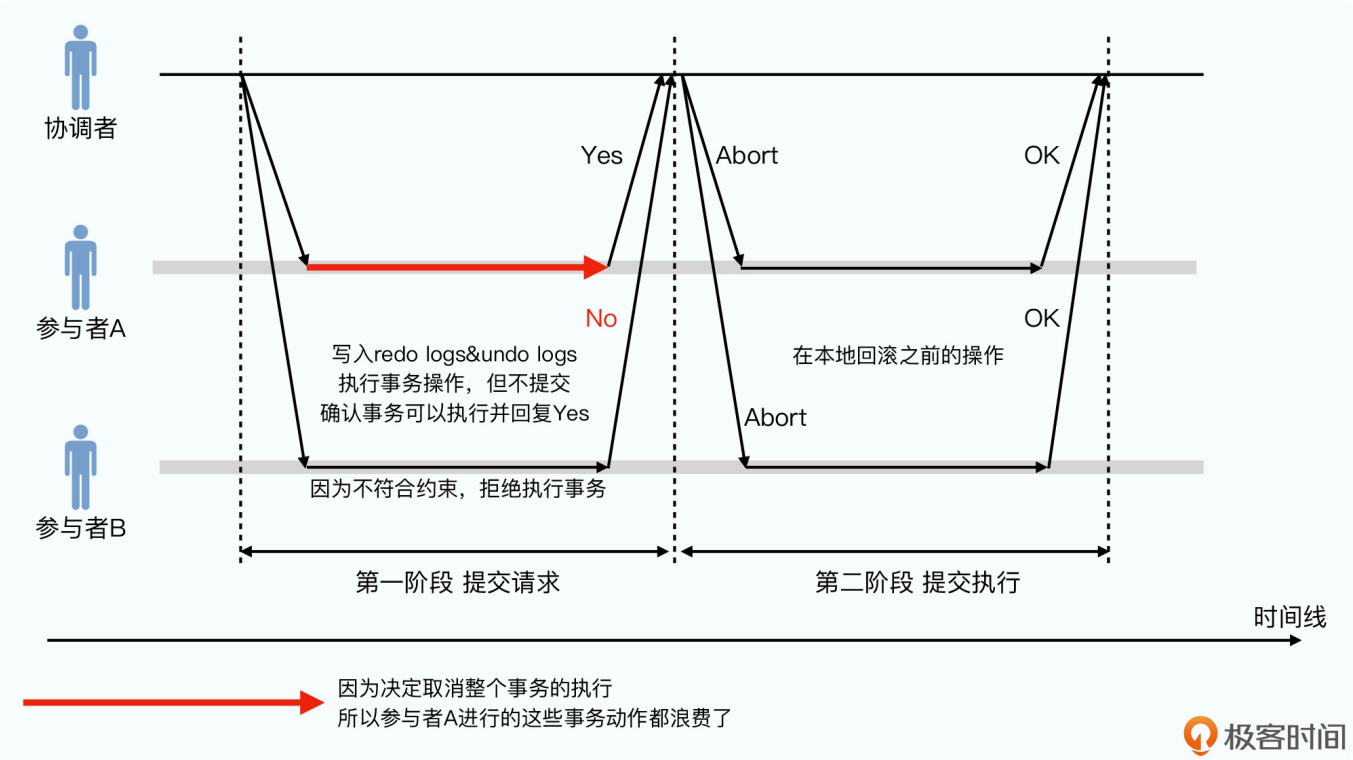


两阶段提交流程

这里需要注意的是，所有的参与者，一旦在提交请求阶段答应自己会执行事务，就不能再反悔了。如果参与者觉得自己不能执行对应的事务，就需要在提交请求阶段就拒绝掉。

比如，如果参与者是一个 MySQL 数据库，那么如果协调者发起的数据写入请求，可能会违背 MySQL 里某个表的字段的唯一性约束。这样 MySQL 数据库就应该在提交请求阶段告诉协调者，而不是等到要实际执行的时候才说。

而协调者这个时候，就会在提交执行阶段，直接发送事务回滚的请求。这个时候，各个参与者写下的 undo logs 就会派上用场了，各个节点可以回滚刚才写入的数据，整个事务也就没有发生。



两阶段提交，任何一个参与者表示无法参与事务时，整个事务会Abort取消掉

如果打一个生活中的比方，这个两阶段提交，就好像我们买卖房子一样，会分成签订合同和实际交房两个阶段。协调者是房屋中介，当他和买卖双方协调完毕，两边都签字确认之后，就不可更改，之后再进行实际交房。

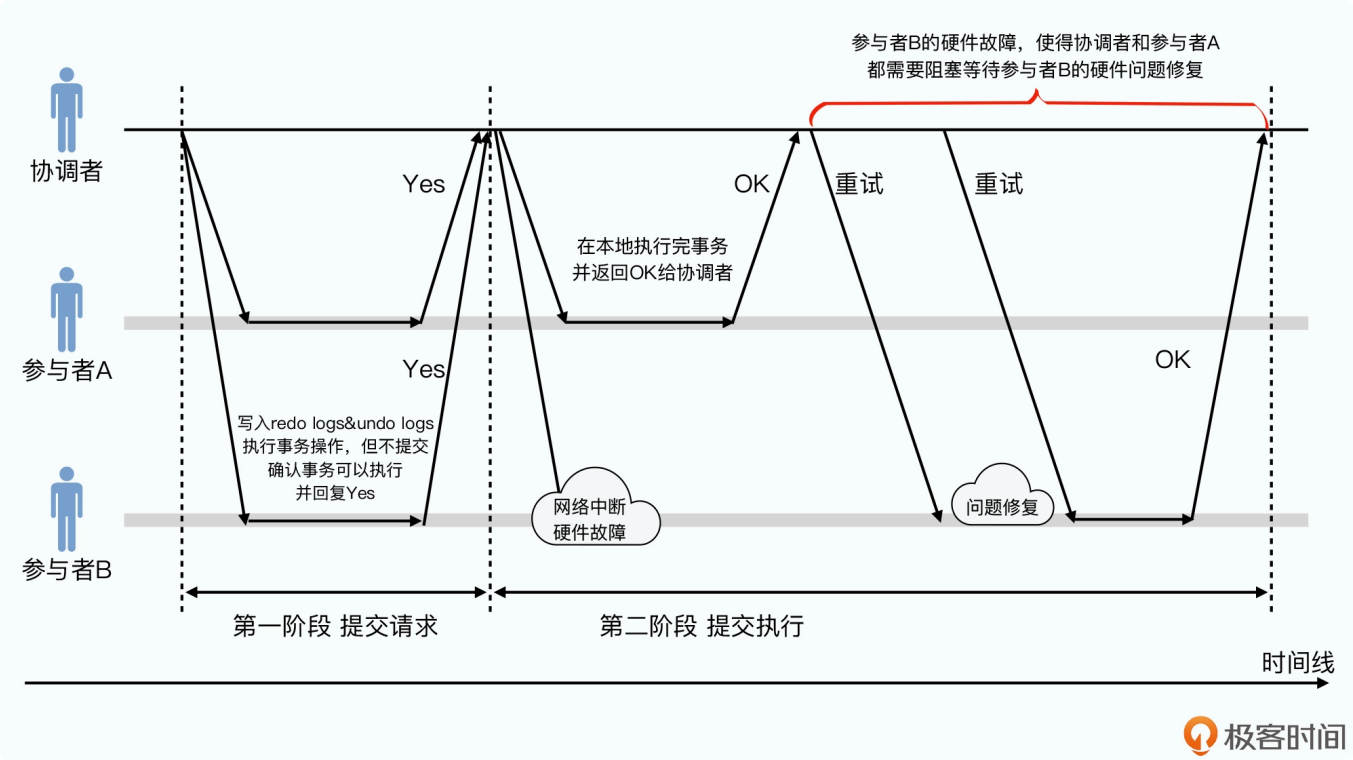
此时此刻，相信聪明的你一定想起了我们之前一直反复会问的一个问题。那就是，**在这个两阶段提交的过程中，如果出现了硬件和网络故障，会发生什么事情呢？**

> 如果是参与者发生了硬件故障，或者参与者和协调者之间的网络出现了故障。这个时候的硬件或者网络故障，就意味着参与者没有办法知道协调者到底想要继续推进事务，还是想要回滚。在这种情况下，参与者在硬件故障解决之后，会一直等待协调者给出下一步指令。

如果协调者之前已经收到了参与者的答应执行事务的响应，那么协调者会一直尝试重新联系参与者。就好像买房合同你已经签了，但是交房时手机没电了，那么房产中介会不断联系你，直到你接起来电话为止。而如果参与者答应执行事务的响应还没有来得及给协调者，那么协调者在等待一段时间没有得到响应之后，会最终决定放弃整个事务，整个事务会回滚。这就好像当房屋交易合同到了你的手里，但是过了一段时间你没有反应。那么房屋中介就自动认为你已经放弃了这笔交易。

那么，这样也就意味着，当硬件出现故障的时候，可能有一个参与者，已经在自己的节点上完成了事务的执行。但是另外一个参与者，可能要过很长一段时间，在硬件和网络恢复之后，才会完成事务。如果这两个参与者是 Master 和 Backup Master，那么在这段时间里，Master 和 Backup Master 之间的数据就是不一致的。

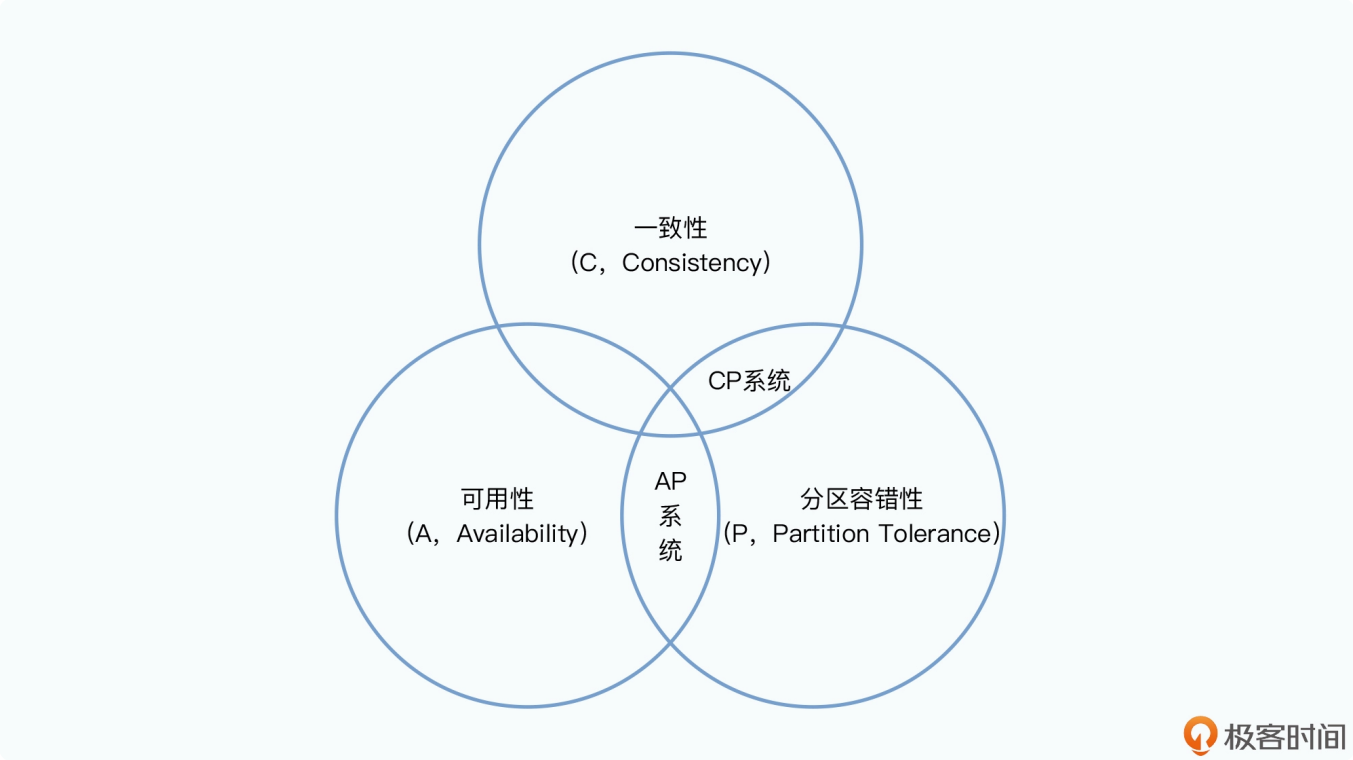
不过，如果外部所有和参与者的沟通，都需要通过协调者的话，协调者完全可以在 Backup Master 还没有恢复的时候，都告知外部的客户端等一等，之前的数据操作还没有完成。



两阶段提交，在单个节点出现问题的时候，整个系统会要阻塞等待，带来较差的可用性

看完前面这个描述，相信你也明白了。在两阶段提交的逻辑里，是通过一个位居中间的协调者来对外暴露接口，并对内确认所有的参与者之间的消息是同步的。不过，两阶段提交的问题也很明显，那就是**两阶段提交虽然保障了一致性（C），但是牺牲了可用性（A）**。无论是协调者，还是任何一个参与者出现硬件故障，整个服务器其实就阻塞住了，需要等待对应的节点恢复过来。

你会发现，两阶段提交里，任何一个服务器节点出问题，都会导致一次“单点故障”。



2PC本质上是一个CP系统，优先考虑一致性，会牺牲可用性

而且，两阶段提交的事务里，选择回滚的事务其实非常浪费。每个节点都要在不知道其他节点究竟是否可以执行事务的情况下，先把完成事务和回滚事务的所有动作都准备好。这个开销可并不小，而且在这个过程中，协调者其实是一直在等待所有参与者给出反馈的。

所以，两阶段提交的分布式事务的**性能往往好不到哪里去**，这个在我们的“大数据”的语境下可不是什么好消息。

三阶段提交和脑裂问题

那么，为了提升整个系统的可用性，有人就会想，要不，我们把提交请求阶段再拆成两步？

第一步，我们不用让各个参与者把执行的动作都准备好，也就是不用去写什么 undo logs 或者 redo logs，而是先判断一下这个事务是不是可以执行，然后再告诉协调者。这一步的请求叫做 **CanCommit 请求**。

第二步，当协调者发现大家都说可以执行的时候，再发送一个预提交请求，在这个请求的过程里，就和两阶段提交的过程中一样。所有的参与者，都会在这个时候去写 redo logs

和 undo logs。这一步的请求呢，叫做 **PreCommit 请求**。

在 CanCommit 请求和 PreCommit 请求阶段，所有参与者都可以告诉协调者放弃事务，整个事务就会回滚。如果出现网络超时之类的问题，整个事务也会回滚。不过，把整个提交请求的阶段拆分成 CanCommit 和 PreCommit 两个动作，**缩短了各个参与者发生同步阻塞的时间**。

原先无论任何一个参与者决定不能执行事务，所有的参与者都会白白先把整个事务的 redo logs 和 undo logs 等操作做完，并且在请求执行阶段还要再做一次回滚。

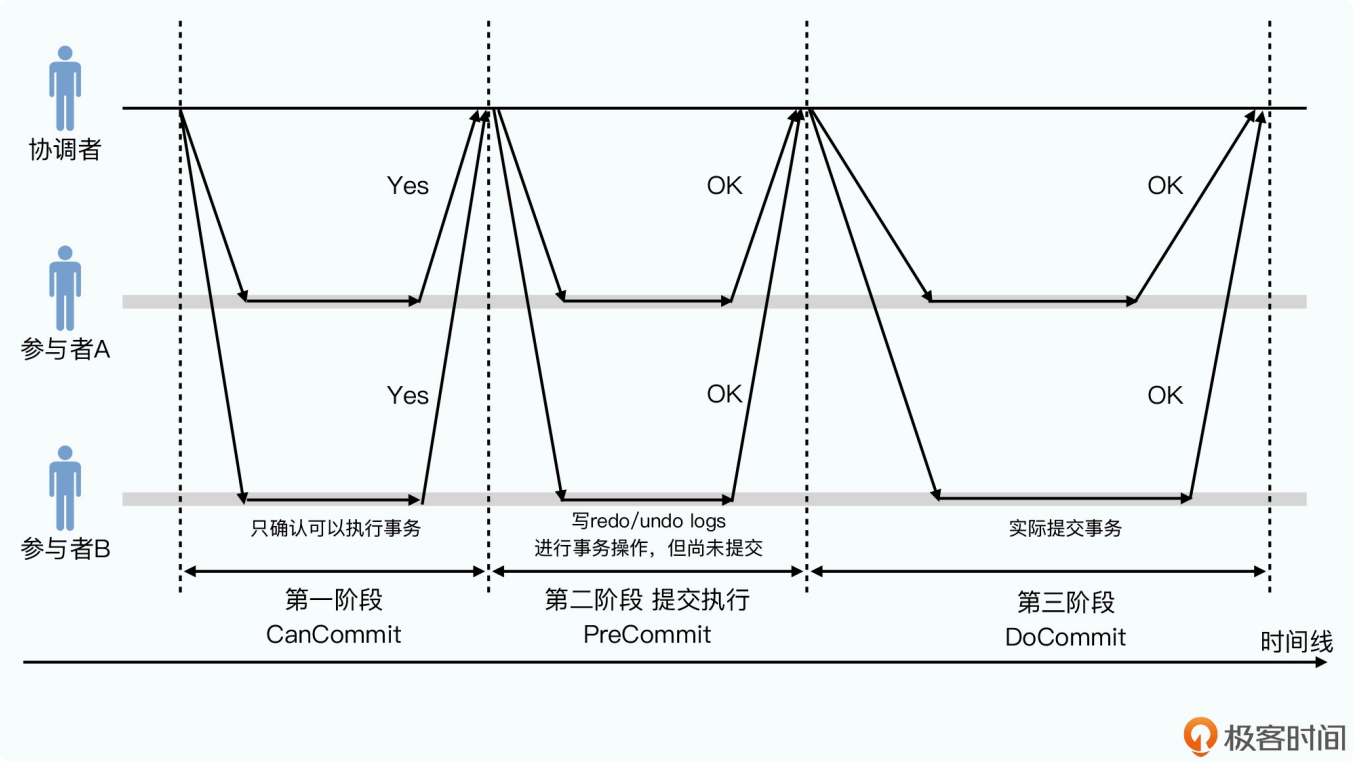
而在新的三阶段提交场景下，大部分不能执行的事务，都可以在 CanCommit 阶段就放弃掉。这意味着所有的参与者都不需要白白做无用功了，也不需要浪费很多开销去写 redo logs 和 undo logs 等等。

另外，在最后的提交执行阶段，三阶段提交为了提升系统的可用性也做了一点小小的改造。在进入最后的提交执行阶段的时候，如果参与者等待协调者超时了，那么参与者不会一直在那里死等，而是会把已经答应的事务执行完成。

这个方式，可以提升整个系统的可用性，在出现一些网络延时、阻塞的情况下，整个事务仍然会推进执行，并最终完成。这个是因为，进入到提交执行阶段的时候，至少所有的参与者已经都在 PreCommit 阶段答应执行事务了。

如果还拿之前的买房来举例，也就是我们把合同的签订拆分成了两个阶段。

第一个阶段是房产中介口头来询问买家和卖家，是不是愿意完成交易。第二个阶段，才是把合同发给双方，去实际签订合同。除非出现意外情况，我们一般不会在第二个阶段反悔合同。



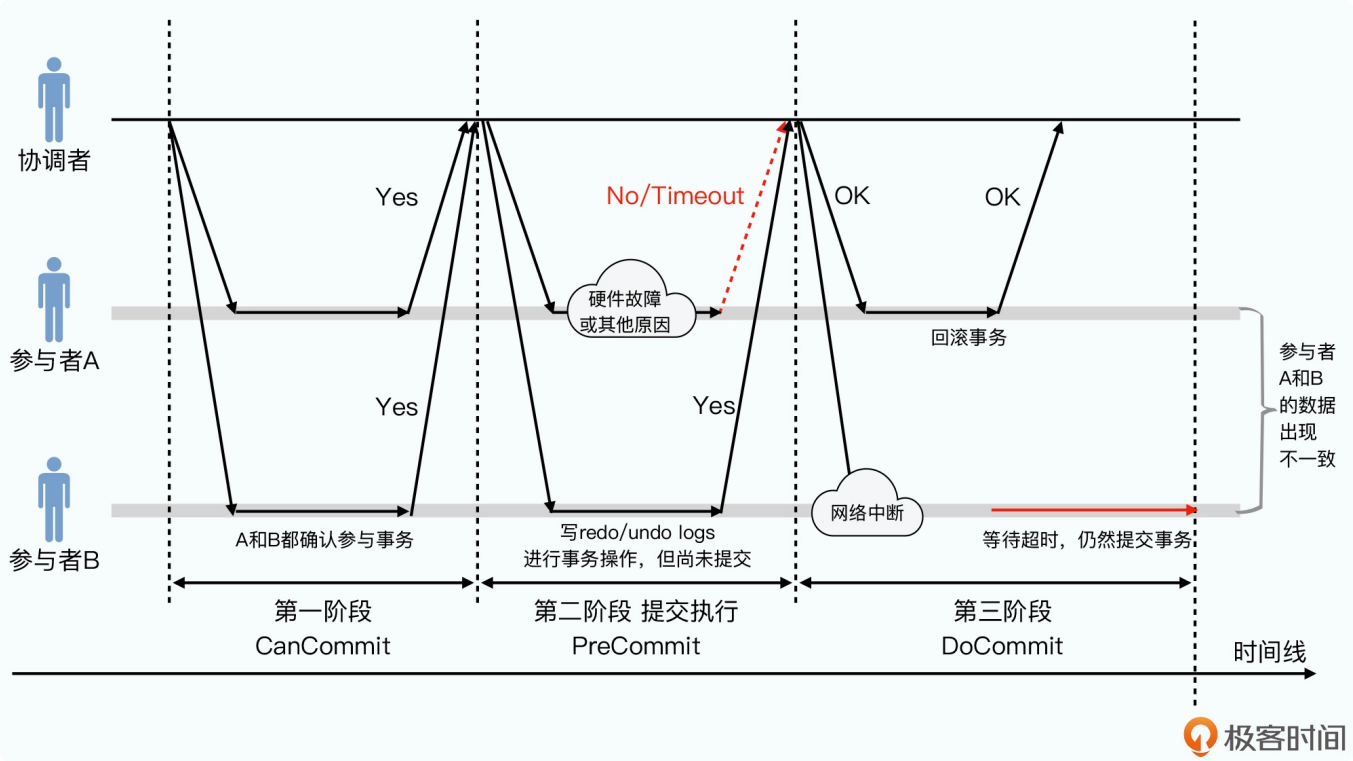
三阶段提交流程图

但是，在一种特殊的情况下，三阶段提交带来的问题会比二阶段更糟糕。这种情况是这样的：

所有参与者在 CanCommit 阶段都答应了执行事务。

在 PreCommit 阶段，协调者发送 PreCommit 信息给所有的参与者之后，参与者 A 挂掉了，所以它没有实际执行事务。协调者收到了这个消息，想要告诉参与者 B。而这个时候，参与者 B 和协调者之间的网络中断了。在等待了一段时间之后，参与者 B 决定继续执行事务。

而在这个时候，就会发生一个很糟糕的状况，那就是参与者 B 的状态和其他的参与者都不一致了。也就是出现了所谓的“**脑裂**”，即系统里不同节点出现了两种不同的状态。



3PC情况下，为了可用性，可能会导致数据不一致，牺牲了一致性

可以看到，**三阶段提交，其实就是在出现网络分区的情况下，仍然尝试执行事务。同时，又为了减少网络分区下，出现数据不一致的情况，选择拆分了提交请求**。把提交请求变成了一个小开销的 CanCommit，和一个大开销的 PreCommit。

这个方法不能不说不好，我们前面指出的那种特殊情况，在传统的数据库事务领域，发生的概率并不高。我们可能只有 2~3 台服务器，每秒发生的事务也并不多。

但是，一旦涉及到“大数据”这三个字，问题又变得不同了。在 Bigtable 的论文讲解里，我们看到的是一个上千台服务器的集群，每秒的数据库读写次数可以上升到百万次。在这个数据量下，所谓的“很少会发生”，就变成了“**必然会发生**”。

实际上，**三阶段提交，就是为了可用性（A），牺牲了一致性（C）**。相信你看到这里，对 CAP 理论应该就找到一些感觉了。

那么，是不是我们就没有更好的办法了呢？

答案当然不是这样的。其实两阶段提交也好，三阶段提交也好，最大的问题并不是在可用性和一致性之间的取舍。而是这两种解决方案，都充满了“**单点故障**”，特别是协调者。

因为系统中有一个中心化的协调者，所以其实整个系统很容易出现“单点故障”。换句话说，就是整个系统的“容错”能力很差。所以，我们需要一个对单个节点没有依赖的策略，即使任何一个单个节点的故障，或者网络中断，都不会使得整个事务无法推进下去。这也是我们下一讲要深入讲解的 **Paxos 算法**。

小结

这一讲里，我们还没有开始解读论文，而是在讲解一些分布式一致性的基础知识，为我们下一讲剖析 Chubby 的论文做好准备。

我们看到，其实本来**我们以为非常简单地同步复制 Master，并不是理所当然的事情**。我们先看了最简单的两阶段提交的策略。通过把事务的执行分成的提交请求和提交执行两个阶段，使得我们可以确保两个节点都一定会执行它们承诺的操作。

但是，两阶段提交也给我们的系统带来了一个新的挑战。也就是在 CAP 中，它选择了一致性（C，Consistency）和分区容错性（P，Partition Tolerance），但是对可用性（A，Availability）做出了妥协。在两阶段提交这个策略下，一旦 Backup Master 节点出现问题，其实整个系统都是不能写入的，换句话说，Backup Master 其实增加了 Master 不可用的概率。

所以为了提升可用性，在两阶段提交之上，就进化出了三阶段提交的算法。但是在提升了可用性的情况下，三阶段提交有可能造成数据不一致，也就是牺牲了一致性。相信学完了这一讲，你对 CAP 理论就有了一个更直观的认识。

两阶段提交和三阶段提交，都会在单个节点出现故障的情况下出现问题。于是，Paxos 算法走上了历史舞台，那么下一讲，我们会来深入讲解分布式事务和 Paxos 算法。

推荐阅读

分布式一致性是一个很有趣但也很烧脑的问题。而为了理解整个 Chubby 系统，你也需要很多这方面的预备知识。我推荐你花一些时间读一下《数据密集型应用系统设计》的 7、8、9 三个章节。我们接下来的一讲也会和这部分知识高度相关。

思考题

最后，给你留一道思考题。在今天讲解的两阶段提交的过程中，我们的 Master 和 Backup Master 都是分布式事务的参与者。那么，在这个过程中，是由谁来充当协调者的呢？

欢迎在留言区分享你的答案和思考，和其他同学共同探讨。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 通过Thrift序列化：我们要预知未来才能向后兼容吗？

精选留言 (3)

写留言

陈迪

2021-10-15

Chubby是到现在为止最难懂的论文。。一些例子和设计决策依据让人不知所云

扭高达□

2021-10-15

思考题: 协调者应该是client端

展开

1

飞翔

2021-10-15

三阶段提交有用在生产上的例子吗

展开